Comparative Study of Pathfinding Algorithms on Custom Grid Maps

ZIRAN WEI YUNSEN XING
ziranw | yunsen @kth.se

January 20, 2026

Abstract

Pathfinding is a fundamental challenge in autonomous systems and robotics, requiring algorithms that efficiently and reliably navigate complex, dynamic environments. This study evaluates the performance of five classical pathfinding algorithms, including Breadth-First Search (BFS), Depth-First Search (DFS), Dijkstra, Greedy Best-First Search (GBFS), and A*, across diverse grid-based environments. In this research, we evaluated algorithmic efficiency, success rate, path quality, and scalability on seven categories of maps including sparse obstacles, dense obstacles, maze like, weighted, swamp like, trap like, and bottleneck like, based on a 35 map grid benchmark suite. We ran controlled simulations with automated logging of success rate, path length, path cost, computation time, and nodes expanded. Results show that heuristic-driven algorithms (A* and GBFS) achieve higher search efficiency and faster computation, whereas cost-based algorithms (Dijkstra and A*) consistently find lower-cost paths in weighted environments. BFS excel in unweighted maps, producing the shortest geometric paths but requiring greater computational effort in complex scenarios. These findings illustrate trade-offs between optimality, efficiency, and scalability, providing guidance for selecting pathfinding algorithms suited to specific environments and application requirements.

Contents

1 Introduction

Pathfinding is a key research topic in robotics and autonomous systems, involving the identification of a feasible path from a start point to a goal point within a map, commonly represented as a grid or a graph [1]. While many pathfinding algorithms have been proposed, including traditional graph search methods such as Breadth First Search and Dijkstra, as well as heuristic search algorithms such as A*, each approach has distinct advantages and limitations depending on the application context.

The structure of the environment significantly affects algorithm performance. For instance, traditional algorithms may perform comparably to heuristic ones in sparse maps, whereas heuristic algorithms typically reduce search time in maze-like or dense obstacle maps. However, comprehensive comparisons across varying map complexities remain limited. This study aims to systematically evaluate five classic algorithms including BFS, DFS, Dijkstra, Greedy Best First Search, and A* on varying custom map scenarios. By combining quantitative metrics such as success rate, path length, computation time, and node expansion with qualitative visual analysis, this research seeks to clarify the strengths and weaknesses of different algorithms and to provide systematic guidance for algorithm selection and optimization [2].

1.1 Theoretical Framework

The pathfinding algorithms evaluated in this study fall into two primary categories: traditional graph search and heuristic search. Traditional methods, including breadth-first search (BFS), depth-first search (DFS), and Dijkstra's algorithm, explore the search space without goal-directed heuristics. BFS operates by exploring nodes layer by layer, a strategy that guarantees the shortest path in unweighted graphs but often incurs high computational costs in large or complex maps due to exhaustive node expansion. Conversely, DFS explores deeply along a single path until a goal or dead end is reached; while this approach can be memory-efficient and faster in specific layouts, it offers no guarantee of optimality and may produce suboptimal routes. Dijkstra's algorithm extends these principles to weighted graphs, ensuring the discovery of the globally optimal path by exploring nodes based on accumulated cost. However, its lack of heuristic guidance necessitates a uniform exploration of all promising nodes, often resulting in heavy computational overhead [3].

In contrast, heuristic search algorithms such as Greedy Best-First Search (GBFS) and A* utilize heuristic functions to estimate the remaining cost to the goal, thereby directing the search more efficiently. GBFS prioritizes nodes that appear closest to the goal, offering rapid convergence but potentially sacrificing path optimality [4]. The A* algorithm refines this approach by combining the actual cost incurred from the start node with the heuristic estimate to the goal. This synthesis allows A* to balance path optimality with search efficiency, maintaining near-optimal solutions while significantly reducing the search space compared to uninformed methods [5].

1.2 Previous Studies

Several recent studies provide a foundation for our research. Prior work [2] compared multiple pathfinding algorithms using the PathBench framework, focusing on path length and computational efficiency. Another study [6] proposed optimizations for pathfinding on weighted maps. Recent work [7] investigated path smoothing techniques for RRT type algorithms.

Despite these contributions, these literature often evaluates algorithms on a limited set of scenarios or focuses narrowly on specific metrics like runtime and path length. This research bridges these gaps by systematically examining five distinct algorithms across seven diverse map types (sparse, dense, maze-like, weighted, swamp-like, trap-like, and bottleneck-like), incorporating both node expansion analysis and visual assessments to provide a holistic evaluation of algorithmic performance.

1.3 Research Questions

To systematically evaluate these algorithms, this study addresses several core inquiries regarding their performance across diverse environments. The primary objective is to determine which algorithm achieves the highest success rate in finding feasible paths across different types of maps. Furthermore, the study investigates path quality by analyzing which algorithm consistently produces the shortest path length and how map structure influences this optimality. Beyond path quality, the research examines computational efficiency by comparing average search times and search space efficiency, measured by the number of nodes expanded. Finally, the study seeks to understand scalability by quantifying how significantly the performance of each algorithm degrades as map complexity increases.

1.4 Hypotheses

Based on the theoretical characteristics of the selected algorithms, this study formulates three hypotheses.

1. Dijkstra's algorithm and A* will consistently produce the lowest cost paths in weighted environments, outperforming Greedy Best First Search.

2. Greedy Best First Search will yield lower average computation times than both Dijkstra's algorithm and A* in sparse obstacle maps.

3. In maze like environments, heuristic driven algorithms such as A* and Greedy Best First Search will expand fewer nodes than uninformed algorithms such as breadth first search and depth first search when locating a goal.

2 Method

This study employed a quantitative experimental approach to systematically evaluate the performance of five classical pathfinding algorithms. All experiments were conducted within a controlled computer simulation platform, ensuring identical execution conditions across diverse grid-based environments. This design facilitated the collection of consistent and statistically comparable performance data. The simulation platform and experimental implementation are publicly available to support reproducibility [8].

2.1 Experimental Design and Variables

The study utilized a factorial design where the primary independent variables were the pathfinding algorithm (five levels) and the map topology (seven types). The *map complexity* served as a secondary independent factor, varied through obstacle density and structural layout within each category.

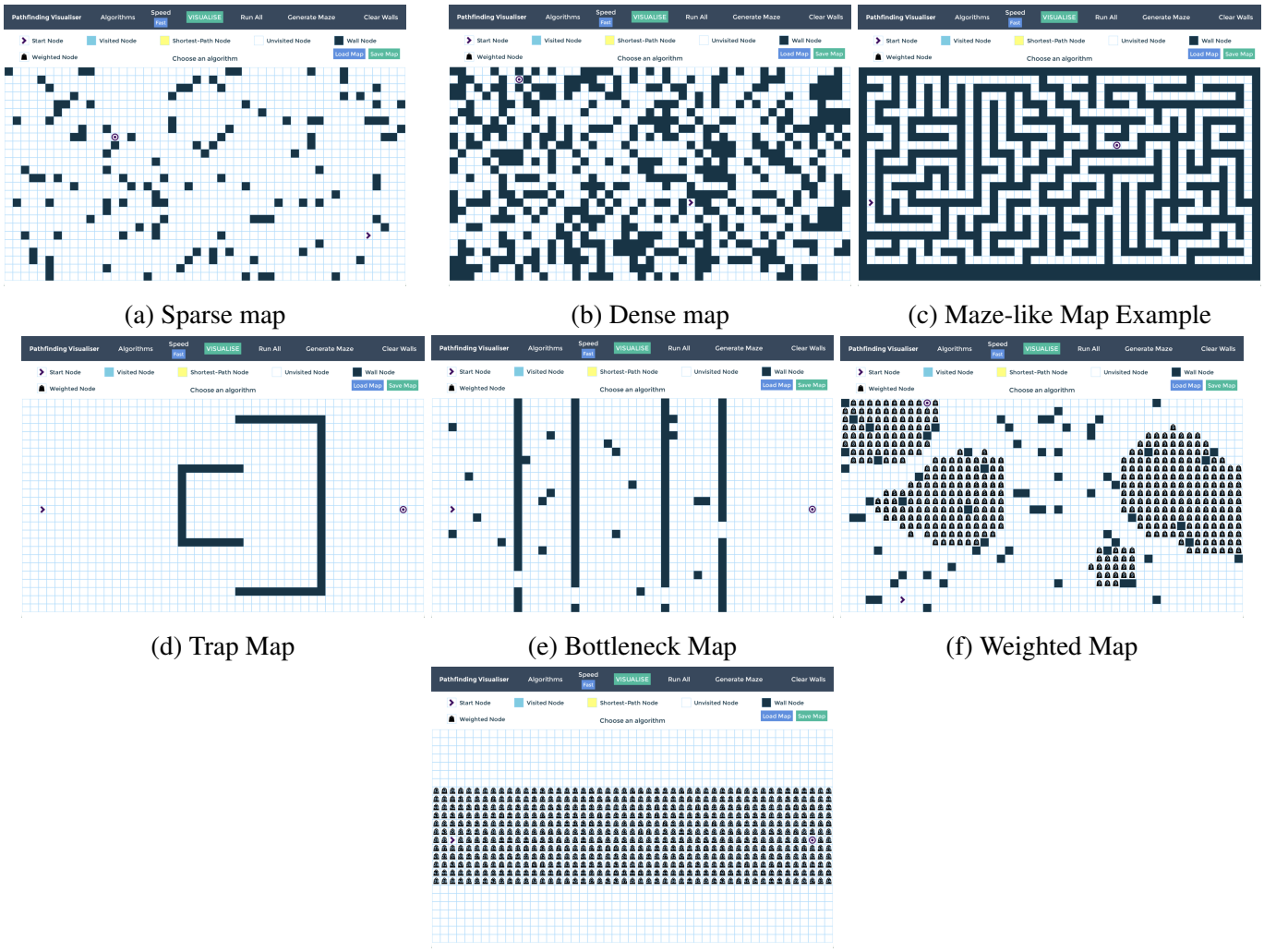
The dependent variables were the quantitative performance metrics measured during each trial: success rate, path length, computation time, and the number of nodes expanded. To control for confounding variables, the start and goal coordinates were fixed for each map instance, and all algorithms were executed on the same hardware configuration to minimize processing variance.

2.2 Benchmark Map Scenarios

To evaluate algorithmic robustness across diverse environments, a benchmark suite of 35 grid maps (49×26 dimension) was constructed. These maps are categorized into seven distinct topological types, each designed to isolate specific navigational challenges.

- **Sparse and Dense Maps:** GGenerated with random obstacle distributions. Sparse maps with an obstacle density of approximately 10% test algorithm speed in open spaces, while Dense

maps featuring an obstacle density of approximately 40% evaluate maneuverability in cluttered environments, see Figure 1a and Figure 1b.



(a) Sparse map (b) Dense map (c) Maze-like Map Example

(d) Trap Map (e) Bottleneck Map (f) Weighted Map

(g) Swamp Map

Figure 1: Demonstration of seven map structures

- **Maze-like Maps:** Constructed using recursive backtracking algorithms to create long, winding corridors with perfect mazes containing no loops or braided mazes. These maps test the algorithm's ability to handle deep recursion and long detours, see Figure 1c.
- **Trap Maps:** Feature concave, U-shaped obstacles ("local minima") placed directly between the start and goal. These are specifically designed to challenge greedy algorithms such as GBFS that may get stuck in dead ends while following heuristics, see Figure 1d.
- **Bottleneck Maps:** Characterized by large open areas separated by a wall with a single, narrow passage measuring 1 to 2 cells wide. This tests whether algorithms can locate specific gateways rather than flooding the entire area, see Figure 1e.
- **Weighted Maps:** Obstacles are replaced by terrains with varying traversal costs ranging from 1 to 100 distributed randomly. These maps differentiate algorithms that optimize for path cost versus those that optimize for step count, see Figure 1f.
- **Swamp Maps:** A variant of weighted maps features large, contiguous regions of high-cost terrain known as the swamp blocking the direct path. An optimal planner should choose to skirt around the swamp, resulting in a longer path with lower cost, whereas a suboptimal planner might cut through it, yielding a shorter path with higher cost, see Figure 1g.

2.3 Simulation Environment and Algorithms

The experiments were performed using a customized version of the Pathfinding-Visualizer platform [9], which was extended to support batch processing and automated data logging. To rigorously test algorithmic robustness, a benchmark suite of 35 grid maps was constructed. These maps were categorized into seven distinct topological types—including sparse obstacles, dense obstacles, maze-like, weighted, swamp-like, trap-like, and bottleneck-like environments—with each category containing five distinct instances to capture structural variations. Five distinct pathfinding algorithms were implemented and evaluated. Their specific search logic and implementation details are described as follows.

- **Breadth-First Search (BFS):** Implemented using a First-In-First-Out (FIFO) queue. The algorithm explores the grid systematically by expanding all neighbor nodes at the current depth before moving to the next level. In our unweighted grid experiments, this guarantees the finding of the shortest path (minimum steps) but requires traversing all nodes uniformly around the start point [10].

- **Depth-First Search (DFS):** Implemented using a Last-In-First-Out (LIFO) stack mechanism (or recursion). The algorithm prioritizes exploring as deep as possible along each branch before backtracking. While memory-efficient, this logic does not guarantee the shortest path and often results in unoptimized, meandering routes in open spaces [11].

- **Dijkstra's algorithm:** Utilizes a priority queue to explore nodes based on the cumulative cost $g(n)$ from the start node, where n denotes the current node in the search graph. It guarantees the optimal path in weighted graphs by always expanding the unvisited node with the lowest known distance. In our implementation, it behaves similarly to uniform cost search, thoroughly exploring all potential paths with lower costs before considering higher cost alternatives [3].

- **A* Algorithm:** Implemented using a priority queue ordered by the total estimated cost $f(n) = g(n) + h(n)$, where $g(n)$ is the actual cost from the start and $h(n)$ is the heuristic estimate to the goal. Similar to GBFS, the Manhattan distance was used as the heuristic function. This logic allows A* to balance the optimality guarantees of Dijkstra with the search speed of GBFS, ensuring the most efficient optimal path is found provided the heuristic is admissible [5].

- **Greedy Best-First Search (GBFS):** A heuristic-driven algorithm that uses a priority queue ordered solely by the heuristic value $f(n) = h(n)$ (where $g(n)$ is ignored), which estimates the cost from the current node n to the goal. In this study, the Manhattan distance was employed as the heuristic function $h(n)$ to align with the 4-directional movement constraint. This approach prioritizes speed by aggressively moving towards the goal but is not guaranteed to find the shortest path [4].

2.4 Data Collection and Evaluation Metrics

To ensure a multifaceted evaluation of algorithmic performance, data collection was fully automated within the simulation framework. Crucially, the grid environment was configured to enforce a 4-directional movement constraint, restricting agents to cardinal movements (North, South, East, West). Consequently, the geometric distance between any two adjacent nodes is strictly defined by the Manhattan geometry. For every experimental trial, the system recorded the following four quantitative metrics.

- **Success Rate (SR):** This binary metric indicates the reliability of the algorithm. It is calculated as the ratio of successful trials ($N_{success}$) where a valid path is found to the total number of trials (N_{total}) for a given scenario

$$SR = \frac{N_{success}}{N_{total}} \times 100\% \tag{1}$$

- **Path Length (L):** Defined as the total accumulated cost of the generated path P. Let P be a sequence of nodes $\langle v_1, v_2, \ldots, v_k \rangle$ from the start node v_1 to the goal node v_k. The path length is computed as the

sum of the edge weights $w(v_i, v_{i+1})$ between consecutive nodes

$$L(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}) \tag{2}$$

- **Path Cost (C):** Defined as the total accumulated traversal cost of the generated path P. Let P be a sequence of nodes $\langle v_1, v_2, \ldots, v_k \rangle$. The path cost is computed as the sum of the traversal costs $c(v_i, v_{i+1})$ associated with moving between consecutive nodes

$$C(P) = \sum_{i=1}^{k-1} c(v_i, v_{i+1}) \tag{3}$$

  In unweighted maps, the cost is uniform ($c = 1$), meaning the path cost C is equivalent to the path length L (number of steps). In *weighted maps*, c varies to represent different terrain difficulties (e.g., $c \in \{1, 5, 10\}$), allowing the metric to differentiate between shorter paths and "cheaper" paths.

- **Nodes Expanded (N_{exp}):** This metric counts the total number of unique grid cells that were visited nodes which are added to the closed set during the search process

$$N_{exp} = |\{v \in V \mid v \text{ is visited}\}| \tag{4}$$

  This serves as a direct proxy for search space efficiency and memory consumption. A strictly lower N_{exp} indicates a more targeted search strategy that efficiently prunes irrelevant areas of the map.

- **Computation Time (T):** The real-time duration required for the algorithm to complete the search, excluding GUI rendering overhead.

2.5 Experimental Procedure

The experimental procedure involved executing every algorithm on each of the 35 maps, resulting in a comprehensive dataset of algorithm-map pairs. We conducted multiple independent trials for each pair to ensure statistical reliability and to mitigate the impact of system level noise such as background process latency.

Prior to the final data collection, a pilot test was performed to verify system stability and ensure that all algorithms terminated correctly under boundary conditions. During the formal experiments, the system automatically logged the performance metrics while simultaneously generating visual traces of the search process. These visualizations were preserved to facilitate qualitative analysis of search behaviors, such as exploration patterns and detour handling, which complement the quantitative data in the subsequent analysis.

3 Results and Analysis

This section presents the quantitative experimental results derived from the 35 benchmark maps. We analyzed the primary metrics to evaluate algorithmic performance. The analysis is structured to assess overall performance, robustness to obstacle density, efficiency in complex structures, and sensitivity to traversal costs.

Table 1: Overall Average Performance (Summary of 35 Maps)

Algorithm	Success Rate (%)	Path Cost	Path Length	Nodes Exp.	Time (ms)
BFS	100	290.6	**64.7**	830	2.08
DFS	100	457.1	183.0	421	5.44
Dijkstra	100	**115.7**	67.4	850	5.18
GBFS	100	296.6	69.7	**162**	**0.89**
A*	100	**115.7**	67.3	479	2.86

Table 2: Impact of obstacle density on average performance (sparse vs. dense).

Algorithm	Sparse Time (ms)	Sparse Nodes	Dense Time (ms)	Dense Nodes
BFS	2.68	934	0.47	278
DFS	9.81	590	0.57	190
Dijkstra	6.19	930	0.99	277
GBFS	0.17	49	0.28	93
A*	0.34	73	0.74	186

3.1 Overall Quantitative Performance

All five algorithms achieved a 100% success rate, successfully identifying a valid path from the start to the goal in every tested scenario, see table 1. However, the quality of the solutions and the computational resources required varied significantly.

As summarized in Table 1, Dijkstra and A* proved to be the optimal planners, achieving the lowest average Path Cost of 115.7. In contrast, GBFS demonstrated the fastest performance with an average time of 0.89 ms, but this speed came at the expense of path quality, resulting in a significantly higher cost of 296.6. DFS exhibited the most erratic behavior, producing the highest average cost of 457.1 and path length of 183.0, which confirms its unsuitability for optimal pathfinding. Notably, A* achieved the best balance, maintaining the same optimal cost of 115.7 as Dijkstra while reducing the search space by approximately 44% from 850 to 479 nodes and cutting computation time by 45% from 5.18 ms to 2.86 ms.

3.2 Impact of Obstacle Density

The density of obstacles significantly influenced algorithmic behavior, but with opposing effects for uniformed versus heuristic algorithms, see Table 2.

In sparse maps, uninformed algorithms such as BFS and Dijkstra expanded a massive number of nodes, reaching approximately 930. This expansion is attributed to the lack of obstacles, which forces the algorithms to exhaustively explore the open space in all directions. Conversely, in dense maps, the search space was significantly reduced; for instance, BFS node expansions dropped from 934 to 278, while Dijkstra saw a similar decrease from 930 to 277. This reduction occurs because obstacles naturally prune the expansion frontier.

Interestingly, heuristic algorithms such as GBFS and A* exhibited the opposite trend. Specifically, GBFS node expansions nearly doubled in dense environments compared to sparse maps, rising from 49 to 93. This increase suggests that while heuristics are highly effective in open spaces, complex obstacle arrangements force the algorithm to deviate from the direct line-of-sight, necessitating local adjustments and increased node expansion.

3.3 Performance in Complex Structures

The experimental results regarding algorithmic efficiency in structurally complex environments such as Trap, Maze, and Bottleneck maps are presented, see Table 3. Unlike the randomly distributed obstacles in Sparse or Dense maps, these topologies introduce deliberate structural constraints designed to challenge

Table 3: Performance in Complex Structures (Average Nodes Expanded)

Algorithm	Trap Map	Maze Map	Bottleneck Map
BFS	1145	464	999
DFS	691	339	378
Dijkstra	1140	464	993
GBFS	**340**	**323**	**236**
A*	485	395	478

Table 4: Cost-Sensitive Performance on Average (Weighted vs. Swamp)

Algorithm	Weighted Cost	Weighted Length	Swamp Cost	Swamp Length
BFS	145.6	**44.4**	1524	**44.0**
DFS	689.0	250.0	1524	**44.0**
Dijkstra	**69.8**	51.2	**376**	56.4
GBFS	154.8	46.4	1524	**44.0**
A*	**69.8**	50.4	**376**	56.4

specific search behaviors such as the ability to escape local minima or navigate constricted passages. By analyzing the number of nodes expanded in these scenarios, we can quantify the actual computational effort required by each algorithm to overcome these navigational difficulties.

The experimental results in Trap maps reveal a stark contrast between uninformed and heuristic-driven search behaviors. BFS and Dijkstra struggled significantly, expanding over 1140 nodes as they were forced to systematically flood the interior of trap structures before finding an exit, see Figure 2a and Figure 2b. In sharp contrast, GBFS demonstrated superior efficiency by expanding only 340 nodes, see Figure 2c. This performance advantage suggests that its aggressive heuristic guidance effectively pulled the search trajectory towards the goal, allowing the algorithm to bypass or quickly exit simple concave obstacles without the exhaustive flooding characteristic of Dijkstra and BFS.

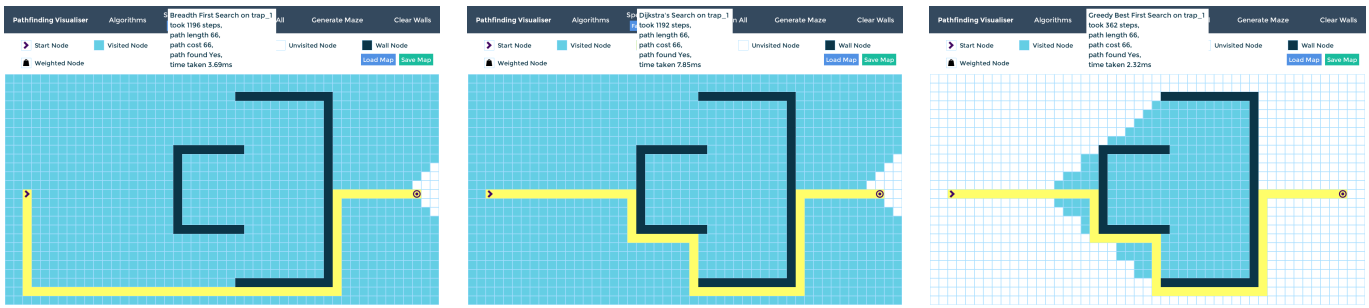
In Maze-like and Bottleneck environments, GBFS maintained its dominance, achieving the lowest node expansion counts of 323 and 236, respectively, which confirms the effectiveness of strong heuristics even in constrained corridors. A notable divergence occurred in the Bottleneck maps, where BFS and Dijkstra expanded nearly 990 nodes by flooding the entire area preceding the narrow passage. However, DFS performed surprisingly well in this scenario, requiring only 378 node expansions; its depth-first nature likely allowed it to penetrate the narrow opening by chance early in the search process, avoiding the need to explore the entire preceding open space.

3.4 Cost-Sensitive Navigation

The distinction between ”shortest” and ”cheaper” paths is most evident in Weighted and Swamp maps, see table 4.

In the Swamp Map scenario, the difference is stark. BFS and GBFS produced a remarkably high Path Cost of 1524 while maintaining a short Path Length of 44.0, see Figure 3a and Figure 3d. This indicates that these algorithms are ”cost-blind” as they selected the geometrically shortest path directly through the high-cost swamp area, oblivious to the terrain difficulty. DFS similarly failed to optimize for cost, likely producing erratic paths with high accumulation of weights due to its depth-first nature, see Figure 3b.

In contrast, Dijkstra and A* produced a much lower Path Cost of 376 despite yielding a longer Path Length of approximately 56.4, see Figure 3c and Figure 3e. This confirms Hypothesis H1 as these algorithms intelligently navigated around the swamp, prioritizing total traversal cost over geometric distance. This result fundamentally validates the necessity of cost-aware algorithms in heterogeneous



(a) BFS on Trap Map (b) Dijkstra on Trap Map (c) GBFS on Trap Map

Figure 2: Algorithmic search behavior in a Trap Map

environments where the shortest path is not necessarily the most efficient one.

(a) BFS (b) DFS (c) Dijkstra

(d) GBFS (e) A*

Figure 3: Algorithmic search behavior on the Swamp Map.

4 Discussion

This study aimed to provide a systematic comparison of pathfinding algorithms across diverse grid-based environments. The experimental results offer quantitative evidence regarding the trade-offs between search optimality, computational speed, and memory efficiency. This section interprets these findings in relation to the initial research hypotheses and discusses the validity of the study.

4.1 Validation of Hypotheses

The empirical data collected from the 35 benchmark maps allows for a direct evaluation of the three hypotheses proposed in Section 1. Regarding the first hypothesis, the results from the Weighted and Swamp maps provide strong support, see Table 4. Dijkstra and A* consistently identified the path with the lowest traversal cost; for example, they achieved a cost of 376 compared to 1524 for BFS and GBFS in Swamp maps. This stark contrast confirms that while BFS and GBFS prioritize geometric distance, Dijkstra and A* effectively optimize for total traversal cost, making them the only viable candidates for cost-sensitive navigation tasks.

The second and third hypotheses regarding computational performance are also supported by the results. Table 2 shows that GBFS was the fastest method in sparse environments, with an average computation time of 0.17 ms. This was significantly lower than the computation time of A*, which was 0.34 ms, and BFS, which was 2.68 ms. These results suggest that the greedy heuristic reduces overhead in open spaces, making GBFS suitable for real time applications where optimality is not the primary objective. The third hypothesis is partially supported by the nodes expanded metric. Table 1 shows that heuristic methods expanded fewer nodes than uninformed methods. GBFS expanded 161 nodes on average, whereas Dijkstra expanded 862 nodes on average. However, DFS outperformed A* in node expansion for Maze and Bottleneck maps, refuting the part of Hypothesis 3 concerning DFS in complex structures. This indicates that while heuristic guidance typically reduces memory usage and limits search space exploration, depth-oriented search can be more efficient in highly constrained environments.

4.2 Reliability and Generalizability

The reliability of these results is underpinned by the controlled experimental design. By enforcing a 4-directional movement constraint and using fixed start/goal configurations, the study ensured that variances in performance were solely attributable to algorithmic differences rather than environmental noise. The use of average values over multiple trials further mitigated the impact of system-level latency.

However, the generalizability of the findings is subject to certain limitations. The experiments were conducted in static, discrete grid environments. Real-world robotics applications often involve continuous spaces, dynamic obstacles, and kinematic constraints such as turning radius, which were not modeled in this study. Consequently, while the relative performance rankings, such as A* being more efficient than Dijkstra, are likely to hold in continuous domains, the absolute performance metrics, including computation time in milliseconds, may differ when applied to high-dimensional configuration spaces.

4.3 Conclusion and Future Work

This research presented a comparative study of five pathfinding algorithms comprising BFS, DFS, Dijkstra, GBFS, and A* across varied map topologies. The results conclusively demonstrate that no single algorithm is superior in all dimensions. A* emerges as the most balanced solution, offering optimality with reasonable computational costs. GBFS excels in speed for time-critical, sparse scenarios but risks severe sub-optimality in complex or weighted terrains. Dijkstra guarantees optimality but suffers from poor scalability due to exhaustive node expansion, while BFS and DFS are generally unsuitable for complex navigation tasks due to their lack of cost-awareness and stability, respectively.

When considering practical deployment scenarios, each algorithm exhibits distinct strengths and limitations. A* is most appropriate when both path optimality and efficiency are required, provided that a well-informed heuristic is available and the environment size and dynamics remain manageable. Its computational cost becomes less acceptable in large-scale or highly dynamic maps where frequent replanning is needed. GBFS is better suited for real-time and time-critical applications, especially in sparse environments or systems with limited computational resources, where rapid path generation is prioritized over optimality. Dijkstra is preferable in weighted graphs when guaranteed optimal solutions are required and computational time is not a primary constraint, although its exhaustive exploration limits scalability. BFS is mainly applicable to small or unweighted environments where uniform step costs are assumed, but it performs poorly in large or complex maps due to excessive node expansion. DFS is generally limited to simple exploration or memory-constrained scenarios, as it lacks optimality guarantees and can produce highly unstable paths in complex navigation tasks.

Future work should expand the scope of this evaluation to address current limitations and enhance its applicability to real-world robotics. First, the movement model should be extended from the current 4-directional Manhattan geometry to 8-directional movement, which allows for diagonal traversal. This would necessitate the evaluation of alternative heuristic functions, such as Octile or Chebyshev distance, to understand their impact on search efficiency. Second, explicit validation of path smoothness is required. By introducing ”jagged” map topologies defined as environments intentionally designed with saw-tooth obstacles, future studies can quantify the ”jaggedness” of generated trajectories. This is critical for determining whether grid-based solutions require post-processing techniques to ensure kinematic feasibility for mobile robots. In addition, the evaluation should be extended to more advanced graph search algorithms, such as Theta* and Jump Point Search, to assess their ability to reduce path length and node expansions while preserving optimality. Finally, to bridge the gap between simulation and high-dimensional control, the evaluation should extend to non-grid environments. This includes testing on Navigation Meshes or implementing sampling-based algorithms such as RRT and PRM in continuous spaces, thereby better representing the challenges of autonomous driving and robotic manipulation.

References

[1] M. M. Costa and M. F. Silva, "A survey on path planning algorithms for mobile robots," in *2019 IEEE International Conference on Autonomous Robot Systems and Competitions (ICARSC)*, 2019. doi: 10.1109/ICARSC.2019.8733623 pp. 1–7.

[2] H.-Y. Hsueh, A.-I. Toma, H. A. Jaafar, E. Stow, R. Murai, P. H. J. Kelly, and S. Saeedi, "Systematic comparison of path planning algorithms using pathbench," *Advanced Robotics*, vol. 36, no. 11, pp. 566–581, 2022. doi: 10.1080/01691864.2022.2062259. [Online]. Available: https://doi.org/10.1080/01691864.2022.2062259

[3] E. DIJKSTRA, "A note on two problems in connexion with graphs." *Numerische Mathematik*, vol. 1, pp. 269–271, 1959. [Online]. Available: http://eudml.org/doc/131436

[4] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. doi: 10.1109/TSSC.1968.300136

[5] S. Erke, D. Bin, N. Yiming, Z. Qi, X. Liang, and Z. Dawei, "An improved a-star based path planning algorithm for autonomous land vehicles," *International Journal of Advanced Robotic Systems*, vol. 17, p. 172988142096226, 10 2020. doi: 10.1177/1729881420962263

[6] M. Carlson, S. K. Moghadam, D. D. Harabor, P. J. Stuckey, and M. Ebrahimi, "Optimal pathfinding on weighted grid maps," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 10, pp. 12 373–12 380, Jun. 2023. doi: 10.1609/aaai.v37i10.26458. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/26458

[7] B. Z. Türkkol, N. Altuntaş, and S. Çekirdek Yavuz, "A smooth global path planning method for unmanned surface vehicles using a novel combination of rapidly exploring random tree and bézier curves," *Sensors*, vol. 24, no. 24, 2024. doi: 10.3390/s24248145. [Online]. Available: https://www.mdpi.com/1424-8220/24/24/8145

[8] W. Ziran and X. Yunsen, "Pathfinding-visualizer (customized version)," https://github.com/AmosWei1995/Pathfinding-Visualizer, 2025, accessed: 2026-01-15.

[9] T. Hilal, "Pathfinding-visualizer," https://github.com/Tauseef-Hilal/Pathfinding-Visualizer, 2021, accessed: 2025-10-07.

[10] C. Y. Lee, "An algorithm for path connections and its applications," *IRE Transactions on Electronic Computers*, vol. EC-10, no. 3, pp. 346–365, 1961. doi: 10.1109/TEC.1961.5219222

[11] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972. doi: 10.1137/0201010. [Online]. Available: https://doi.org/10.1137/0201010

Algorithm 1 Serial-DFS(G, v_0)

```
for each vertex u ∈ V(G) do
    u.visited ← false
end for
STACK ← [v_0]
v_0.visited ← true
while STACK ≠ ∅ do
    u ← POP(STACK)
    for each v ∈ V(G) such that (u, v) ∈ E(G) do
        if not v.visited then
            v.visited ← true
            PUSH(STACK, v)
        end if
    end for
end while
```

Algorithm 2 Serial-BFS(G, v_0)

```
for each vertex u ∈ V(G) − {v_0} do
    u.dist ← ∞
end for
v_0.dist ← 0
Q ← {v_0}
while Q ≠ ∅ do
    u ← DEQUEUE(Q)
    for each v ∈ V(G) such that (u, v) ∈ E(G) do
        if v.dist = ∞ then
            v.dist ← u.dist + 1
            ENQUEUE(Q, v)
        end if
    end for
end while
```

Algorithm 3 A* Search(G, v_0, v_{goal})

for each vertex $u \in V(G)$ **do**
    $u.g \leftarrow \infty$ ▷ Cost from start
    $u.f \leftarrow \infty$ ▷ Estimated total cost
    u.visited $\leftarrow$ false
end for
$v_0.g \leftarrow 0$
$v_0.f \leftarrow h(v_0)$
OPEN $\leftarrow \{v_0\}$
CLOSED $\leftarrow \emptyset$
while OPEN $\neq \emptyset$ **do**
    $u \leftarrow \arg\min_{x \in \text{OPEN}} x.f$
    Remove u from OPEN
    Add u to CLOSED
    if $u = v_{\text{goal}}$ **then**
        return reconstructed path
    end if
    for each v such that $(u, v) \in E(G)$ **do**
        if $v \in$ CLOSED **then**
            continue
        end if
        $g_{\text{new}} \leftarrow u.g + w(u, v)$
        if $g_{\text{new}} < v.g$ **then**
            $v.g \leftarrow g_{\text{new}}$
            $v.f \leftarrow v.g + h(v)$
            v.parent $\leftarrow u$
            if $v \notin$ OPEN **then**
                Add v to OPEN
            end if
        end if
    end for
end while
return failure

Algorithm 4 Greedy Best-First Search(G, v_0, v_{goal})

for each vertex $u \in V(G)$ **do**
    u.visited $\leftarrow$ false
    u.parent $\leftarrow$ None
end for
OPEN $\leftarrow \{v_0\}$
CLOSED $\leftarrow \emptyset$
v_0.visited $\leftarrow$ true
while OPEN $\neq \emptyset$ **do**
    $u \leftarrow \arg\min_{x \in \text{OPEN}} h(x)$
    Remove u from OPEN
    Add u to CLOSED
    if $u = v_{\text{goal}}$ **then**
        return reconstructed path
    end if
    for each v such that $(u, v) \in E(G)$ **do**
        if $v \in$ CLOSED **then**
            continue
        end if
        if not v.visited **then**
            v.visited $\leftarrow$ true
            v.parent $\leftarrow u$
            Add v to OPEN
        end if
    end for
end while
return failure

Algorithm 5 Dijkstra's Search(G, v_0)

for each vertex $u \in V(G)$ **do**
    $u.g \leftarrow \infty$ ▷ Distance from start
    u.visited $\leftarrow$ false
    u.parent $\leftarrow$ None
end for
$v_0.g \leftarrow 0$
OPEN $\leftarrow \{v_0\}$
CLOSED $\leftarrow \emptyset$
while OPEN $\neq \emptyset$ **do**
    $u \leftarrow \arg\min_{x \in \text{OPEN}} x.g$
    Remove u from OPEN
    Add u to CLOSED
    for each v such that $(u, v) \in E(G)$ **do**
        if $v \in$ CLOSED **then**
            continue
        end if
        $g_{\text{new}} \leftarrow u.g + w(u, v)$
        if $g_{\text{new}} < v.g$ **then**
            $v.g \leftarrow g_{\text{new}}$
            v.parent $\leftarrow u$
            if $v \notin$ OPEN **then**
                Add v to OPEN
            end if
        end if
    end for
end while
return all shortest paths from v_0

A Pathfinding Algorithm Pseudocode

B Visual Comparison of Algorithms on Benchmark Maps

Figure 1–5: Results of Five Pathfinding Algorithms on the Bottleneck Map

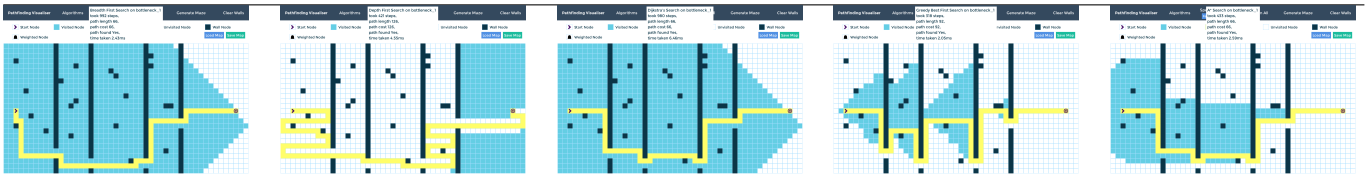


Figure 6–10: Results of Five Pathfinding Algorithms on the dense Map

Figure 11–15: Results of Five Pathfinding Algorithms on the maze Map

Figure 16–20: Results of Five Pathfinding Algorithms on the sparse Map

Figure 21–25: Results of Five Pathfinding Algorithms on the swamp Map

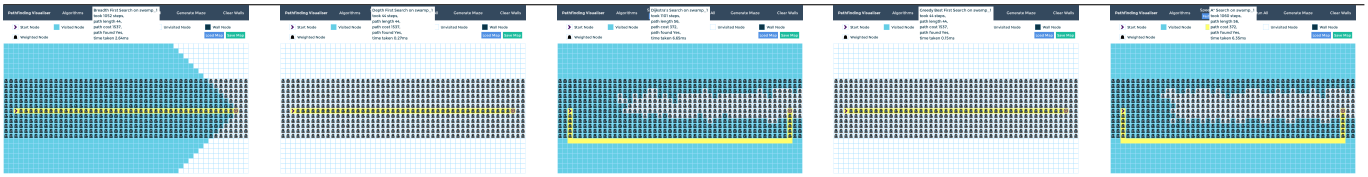


Figure 26–30: Results of Five Pathfinding Algorithms on the trap Map

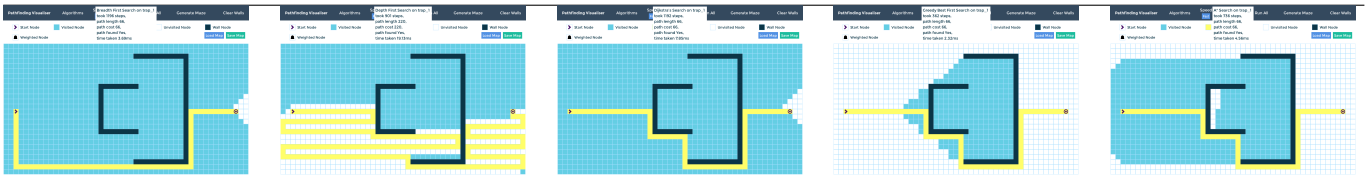
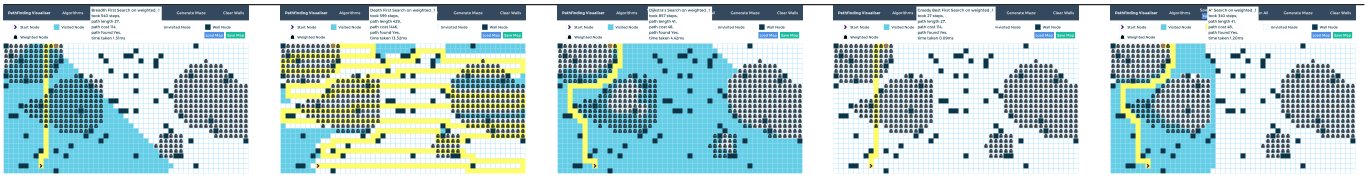


Figure 31–35: Results of Five Pathfinding Algorithms on the weighted Map



C Complete Raw Data

Table 5: Individual Map Results: Bottleneck Maps

Map	Algorithm	Nodes	Length	Cost	Time (ms)
bottleneck_1	BFS	992	66	66.0	2.43
	DFS	421	126	126.0	4.55
	Dijkstra	980	66	66.0	6.46
	GBFS	318	92	92.0	2.05
	A*	433	66	66.0	2.59
bottleneck_2	BFS	991	50	50.0	2.62
	DFS	479	74	74.0	3.93
	Dijkstra	976	50	50.0	6.39
	GBFS	58	52	52.0	0.22
	A*	115	50	50.0	0.48
bottleneck_3	BFS	1016	78	78.0	2.33
	DFS	256	182	182.0	2.40
	Dijkstra	1017	78	78.0	7.06
	GBFS	223	88	88.0	0.95
	A*	588	78	78.0	3.55
bottleneck_4	BFS	1020	84	84.0	2.37
	DFS	387	196	196.0	3.83
	Dijkstra	1022	84	84.0	6.01
	GBFS	373	106	106.0	2.76
	A*	806	84	84.0	5.96
bottleneck_5	BFS	974	66	66.0	2.37
	DFS	346	138	138.0	2.95
	Dijkstra	971	66	66.0	6.31
	GBFS	208	74	74.0	1.15
	A*	446	66	66.0	2.34

Table 6: Individual Map Results: Dense Maps

Map	Algorithm	Nodes	Length	Cost	Time (ms)
dense_1	BFS	299	46	46.0	0.45
	DFS	263	92	92.0	0.81
	Dijkstra	298	46	46.0	0.98
	GBFS	70	52	52.0	0.17
	A*	153	46	46.0	0.48
dense_2	BFS	287	48	48.0	0.47
	DFS	68	52	52.0	0.17
	Dijkstra	286	48	48.0	0.92
	GBFS	105	52	52.0	0.33
	A*	216	48	48.0	0.76
dense_3	BFS	204	42	42.0	0.38
	DFS	151	54	54.0	0.32
	Dijkstra	204	42	42.0	0.61
	GBFS	76	46	46.0	0.25
	A*	112	42	42.0	0.37
dense_4	BFS	471	83	83.0	0.71
	DFS	364	157	157.0	1.35
	Dijkstra	470	83	83.0	2.16
	GBFS	177	85	85.0	0.54
	A*	370	83	83.0	1.81
dense_5	BFS	130	36	36.0	0.32
	DFS	106	50	50.0	0.21
	Dijkstra	126	36	36.0	0.27
	GBFS	38	36	36.0	0.10
	A*	77	36	36.0	0.28

Table 7: Individual Map Results: Maze Maps

Map	Algorithm	Nodes	Length	Cost	Time (ms)
$maze_1$	BFS	567	219	219.0	0.77
	DFS	309	243	243.0	0.52
	Dijkstra	567	219	219.0	2.45
	GBFS	405	219	219.0	1.59
	A*	553	219	219.0	2.38
$maze_2$	BFS	424	83	83.0	0.62
	DFS	351	163	163.0	0.52
	Dijkstra	422	83	83.0	1.55
	GBFS	176	83	83.0	0.47
	A*	218	83	83.0	0.66
$maze_3$	BFS	500	101	101.0	0.64
	DFS	269	215	215.0	0.44
	Dijkstra	498	101	101.0	1.74
	GBFS	314	103	103.0	1.06
	A*	446	101	101.0	2.16
$maze_4$	BFS	437	205	205.0	0.83
	DFS	395	205	205.0	0.55
	Dijkstra	437	205	205.0	1.39
	GBFS	427	205	205.0	1.34
	A*	437	205	205.0	1.62
$maze_5$	BFS	394	99	99.0	0.57
	DFS	371	205	205.0	0.61
	Dijkstra	394	99	99.0	1.40
	GBFS	291	153	153.0	0.93
	A*	321	99	99.0	1.11

Table 8: Individual Map Results: Sparse Maps

Map	Algorithm	Nodes	Length	Cost	Time (ms)
$sparse_1$	BFS	899	43	43.0	2.42
	DFS	353	285	285.0	5.15
	Dijkstra	897	43	43.0	5.08
	GBFS	53	53	53.0	0.19
	A*	104	43	43.0	0.54
$sparse_2$	BFS	1135	53	53.0	3.16
	DFS	472	363	363.0	10.75
	Dijkstra	1135	53	53.0	8.18
	GBFS	60	59	59.0	0.18
	A*	71	53	53.0	0.30
$sparse_3$	BFS	953	43	43.0	3.04
	DFS	837	231	231.0	11.16
	Dijkstra	938	43	43.0	7.12
	GBFS	47	45	45.0	0.15
	A*	68	43	43.0	0.32
$sparse_4$	BFS	1076	47	47.0	3.31
	DFS	761	297	297.0	13.43
	Dijkstra	1076	47	47.0	7.75
	GBFS	55	51	51.0	0.20
	A*	68	47	47.0	0.32
$sparse_5$	BFS	607	28	28.0	1.46
	DFS	525	392	392.0	8.58
	Dijkstra	605	28	28.0	2.82
	GBFS	30	30	30.0	0.12
	A*	56	28	28.0	0.21

Table 9: Individual Map Results: Swamp Maps

Map	Algorithm	Nodes	Length	Cost	Time (ms)
$swamp_1$	BFS	1052	44	1537.0	2.64
	DFS	44	44	1537.0	0.27
	Dijkstra	1101	56	372.0	6.65
	GBFS	44	44	1537.0	0.15
	A*	1060	56	372.0	6.35
$swamp_2$	BFS	1052	44	1488.0	2.85
	DFS	44	44	1488.0	0.25
	Dijkstra	1182	58	381.0	7.83
	GBFS	44	44	1488.0	0.16
	A*	1141	58	381.0	7.17
$swamp_3$	BFS	1052	44	1477.0	2.65
	DFS	44	44	1477.0	0.27
	Dijkstra	1080	56	365.0	6.49
	GBFS	44	44	1477.0	0.16
	A*	1032	56	365.0	6.17
$swamp_4$	BFS	1052	44	1575.0	2.89
	DFS	44	44	1575.0	0.24
	Dijkstra	1147	56	385.0	6.96
	GBFS	44	44	1575.0	0.14
	A*	1099	56	385.0	6.68
$swamp_5$	BFS	1052	44	1543.0	2.65
	DFS	44	44	1543.0	0.24
	Dijkstra	1119	56	376.0	6.75
	GBFS	44	44	1543.0	0.15
	A*	1073	56	376.0	7.48

Table 10: Individual Map Results: Trap Maps

Map	Algorithm	Nodes	Length	Cost	Time (ms)
$trap_1$	BFS	1196	66	66.0	3.69
	DFS	901	220	220.0	19.13
	Dijkstra	1192	66	66.0	7.85
	GBFS	362	66	66.0	2.32
	A*	736	66	66.0	4.56
$trap_2$	BFS	1177	60	60.0	3.51
	DFS	468	284	284.0	4.69
	Dijkstra	1174	60	60.0	9.83
	GBFS	300	60	60.0	2.45
	A*	326	60	60.0	1.86
$trap_3$	BFS	1136	50	50.0	3.73
	DFS	487	256	256.0	7.63
	Dijkstra	1126	50	50.0	7.05
	GBFS	54	50	50.0	0.23
	A*	158	50	50.0	0.49
$trap_4$	BFS	1008	64	64.0	2.27
	DFS	685	238	238.0	6.65
	Dijkstra	1004	64	64.0	5.96
	GBFS	609	64	64.0	7.73
	A*	648	64	64.0	4.08
$trap_5$	BFS	1207	62	62.0	3.23
	DFS	916	216	216.0	16.08
	Dijkstra	1205	62	62.0	9.04
	GBFS	376	62	62.0	2.01
	A*	555	62	62.0	2.54

Table 11: Individual Map Results: Weighted Maps

Map	Algorithm	Nodes	Length	Cost	Time (ms)
$weighted_1$	BFS	540	27	114.0	1.31
	DFS	599	429	1446.0	13.32
	Dijkstra	857	41	48.0	4.42
	GBFS	27	27	114.0	0.09
	A*	340	41	48.0	1.20
$weighted_2$	BFS	846	38	105.0	2.09
	DFS	1131	56	123.0	17.60
	Dijkstra	986	48	74.0	5.60
	GBFS	40	40	109.0	0.12
	A*	608	44	74.0	2.92
$weighted_3$	BFS	1086	53	235.0	2.55
	DFS	413	257	745.0	7.01
	Dijkstra	955	57	57.0	5.48
	GBFS	55	55	237.0	0.19
	A*	463	57	57.0	3.05
$weighted_4$	BFS	1030	50	176.0	2.58
	DFS	629	348	799.0	14.11
	Dijkstra	1146	50	102.0	7.34
	GBFS	56	56	216.0	0.18
	A*	1014	50	102.0	6.96
$weighted_5$	BFS	1171	54	98.0	2.93
	DFS	798	160	332.0	10.66
	Dijkstra	1150	60	68.0	7.56
	GBFS	54	54	98.0	0.16
	A*	864	60	68.0	10.25